7.1 The stdout and stdin file pointers

The stdout file pointer

Programs often need to output data to a screen, file, or elsewhere. A ***FILE**, called a "file pointer," is a pointer to a FILE structure that allows programs to read and write to files. FILE* is available via `#include <stdio.h>`.[Pointers]

The FILE structure maintains the information needed to access files. The FILE structure typically maintains an output buffer that temporarily stores characters until the system copies those characters to disk or screen.

stdout is a predefined FILE* that is pre-associated with a system's standard output, usually a computer screen. The following animation illustrates.

PARTICIPATION ACTIVITY 7.1.1: Writing to stdout using fprintf().

Animation captions:

1. The fprintf() function converts the string literal to characters, temporarily storing characters in an output buffer.
2. The system then writes the buffer's content to screen.

The fprintf() function, or "file print", writes a sequence of characters to a file. The first argument to fprintf() is the FILE* to the file being written. The remaining arguments for fprintf() work the same way as the arguments for printf().

The second argument for the fprintf() function is the ***format string*** that specifies the format of the text that will be printed along with any number of ***format specifiers*** for printing values stored in variables. The arguments following the format string are the expressions to be printed for each of the format specifiers within the format string.

Basic use of printf() and format specifiers was covered in an earlier section, and can be used similarly for fprintf().

PARTICIPATION ACTIVITY 7.1.2: fprintf() and stdout.

1) Write a statement using fprintf() that prints "Enter your age: " to stdout.

Check Show answer

2) Write a statement using fprintf() to print an int variable named numSeats to stdout.

Check Show answer

3) Write a statement using fprintf() to print two float variables named x and y separated by a single comma to stdout.

Check Show answer

4) Will the following two statements both print the same result to the standard output (answer Yes or No)?

```
fprintf(stdout, "nums:");
printf("nums:");
```

Check Show answer

The stdin file pointer

Programs need a way to receive input data, whether from a keyboard, touchscreen, or elsewhere. The ***fscanf()*** function is used to read a sequence of characters from a file, storing the converted values into the specified variables; the first "f" stands for "file." The first argument to fscanf() is a FILE* to the file being read. The remaining arguments for fscanf() work the same way as the arguments for scanf().

The second argument for the fscanf() function is the ***format string*** that specifies the type of value to be read using a ***format specifier***. The argument following the format string is the location to store the value that is read.

stdin is a predefined FILE* (a file pointer[FilePointer]) that is pre-associated with a system's standard input, usually a computer keyboard. The system automatically puts the standard input into a data buffer associated with stdin, from which fscanf() can extract data. The following animation illustrates.

PARTICIPATION ACTIVITY 7.1.3: Reading from stdin using fscanf().

Animation captions:

1. The system puts the standard input into a data buffer associated with stdin.
2. The fscanf() function reads characters from the data buffer up to the next whitespace, converts to the target variable's data type, and stores the result into the variable.

Basic use of scanf() and format specifiers were covered in an earlier section, and can similarly be used for fscanf().

PARTICIPATION ACTIVITY 7.1.4: fscanf() and scanf().

1) Write a statement using fscanf() to read an integer value from stdin, storing the value within an int variable named maxEntries.

Check Show answer

2) Write a statement using fscanf() to read a floating-point value from stdin, storing the value within a float variable named tempSetPoint.

Check Show answer

3) Will the following two statements both read a single integer from the standard input (answer Yes or No)?

```
fscanf(stdin, "%d", &x);
scanf("%d", &x);
```

Check Show answer

Exploring further:

- More on stdin, stdout, and stderr from msdn.microsoft.com
- stdin Reference Page from cplusplus.com

(*Pointers) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.

(*FilePointer) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.

7.2 Output formatting

A programmer can adjust the way that a program's output appears, a task known as ***output formatting***. The format specifiers within the format string of printf() and fprintf() can include ***format sub-specifiers***. These sub-specifiers specify how a value stored within a variable will be printed in place of a format specifier.

Floating-point values

Formatting floating-point output is commonly done using sub-specifiers. A ***sub-specifier*** provides formatting options for a format specifier and are included between the % and format specifier character. Ex: The .1 sub-specifier in `printf("%.1f", myFloat);` causes the floating-point variable myFloat to be output with only 1 digit after the decimal point; if myFloat is 12.34, the output would be 12.3. Format specifiers and sub-specifiers use the following form:

Construct 7.2.1: Format specifiers and sub-specifiers.

```
%(flags)(width)(.precision)specifier
```


The table below summarizes the floating-point sub-specifiers available. Assume myFloat has a value of 12.34. Recall that "%f" is used for float values, "%lf" is used for double values, "%e" is used to display float values in scientific notation, and "%le" is used to display double values in scientific notation.

Table 7.2.1: Floating-point formatting.

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7.2f", myFloat);` `Value:   12.34`

.precision	Specifies the number of digits to print following the decimal point. If the precision is not specified, a default precision of 6 is used.	`printf("%.4f", myFloat);`
		`12.3400`
		
		`printf("%3.4e", myFloat);`
		`1.2340e+01`

flags	-: Left aligns the output given the specified width, padding the output with spaces.
	+: Prints a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+f", myFloat);`
		`+12.340000`
		`printf("%08.2f", myFloat);`
		`00012.34`

Figure 7.2.1: Example output formatting for floating-point numbers.


```
Enter a distance in miles: 10.3
10.30 miles would take:
0.02 hours to fly
0.17 hours to drive
```

```c
#include <stdio.h>

int main(void) {
   double miles;      // User defined distance
   double hoursFly;   // Time to fly distance
   double hoursDrive; // Time to drive distance

   // Prompt user for distance
   printf("Enter a distance in miles: ");
   scanf("%lf", &miles);

   // Calculate the correspond time to fly/drive distance
   hoursFly = miles / 500.0;
   hoursDrive = miles / 60.0;

   // Output resulting values
   printf("%.2lf miles would take:\n", miles);
   printf("%.2lf hours to fly\n", hoursFly);
   printf("%.2lf hours to drive\n\n", hoursDrive);

   return 0;
}
```


PARTICIPATION ACTIVITY 7.2.1: Formatting floating point outputs using printf().

What is the output from the following print statements, assuming

```c
float myFloat = 45.1342f;
```

1) `printf("%09.3f", myFloat);`

Check Show answer

2) `printf("%.3e", myFloat);`

Check Show answer

3) `printf("%09.2f", myFloat);`

Check Show answer

Integer values

Formatting of integer values is also done using sub-specifiers. The integer sub-specifiers are similar to the floating-point sub-specifiers except no .precision exists. For the table below, assume myInt is 301.

Table 7.2.2: Integer formatting.

Sub-specifier	Description	Example
width	Specifies the minimum number of characters to print. If the formatted value has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces (or 0's if the '0' flag is specified).	`printf("Value: %7d", myInt);` `Value:  301`

flags	-: Left aligns the output given the specified width, padding the output with spaces.
	+: Print a preceding + sign for positive values. Negative numbers are always printed with the - sign.
	0: Pads the output with 0's when the formatted value has fewer characters than the width.
	space: Prints a preceding space for positive value.
		`printf("%+d", myInt);`
		`+301`
		
		`printf("%08d", myInt);`
		`00000301`
		
		`printf("%+08d", myInt);`
		`+0000301`

Figure 7.2.2: Output formatting for integers.

```c
#include <stdio.h>

int main(void) {
   const unsigned long KM_EARTH_TO_SUN = 149598000;        // Dist from Earth to sun
   const unsigned long long KM_PLUTO_TO_SUN = 5906376272;  // Dist from Pluto to sun

   // Output distances with min number of characters
   printf("Earth is %11lu", KM_EARTH_TO_SUN);
   printf(" kilometers from the sun.\n");
   printf("Pluto is %11llu", KM_PLUTO_TO_SUN);
   printf(" kilometers from the sun.\n");

   return 0;
}
```



```
Earth is   149598000 kilometers from the sun.
Pluto is  5906376272 kilometers from the sun.
```

PARTICIPATION ACTIVITY 7.2.2: Formatting integer outputs using printf().

What is the output from the following print statements, assuming

```c
int myInt = -713;
```

1) `printf("%+04d", myInt);`

Check Show answer

2) `printf("%05d", myInt);`

Check Show answer

3) `printf("%+02d", myInt);`

Check Show answer

Strings

Strings may be formatted using sub-specifiers. For the table below, assume the myString variable is "Formatting".

Table 7.2.3: String formatting.

Sub-specifier	Description	Example

width	Specifies the minimum number of characters to print. If the string has more characters than the width, the value will not be truncated. If the formatted value has fewer characters than the width, the output will be padded with spaces.	
		`printf("%20s String", myString);`
		`Formatting String`

.precision	Specifies the maximum number of characters to print. If the string has more characters than the precision, the string will be truncated.	`printf("%.6s", myString);` `Format`
flags	-: Left aligns the output given the specified width, padding the output with spaces.	`printf("%-20s String", myString);` `Formatting String`

Figure 7.2.3: Example output formatting for Strings.

```
#include <stdio.h>

int main(void) {

   printf("Dog age in human years (dogyears.com)\n\n");
   printf("-------------------------\n");

   // set num char for each column, left aligned
   printf("%-10s | %-12s\n", "Dog age", "Human age");
   printf("-------------------------\n");

   // set num char for each column, first col left aligned
   printf("%-10s | %12s\n", "2 months", "14 months");
   printf("%-10s | %12s\n", "6 months", "5 years");
   printf("%-10s | %12s\n", "8 months", "9 years");
   printf("%-10s | %12s\n", "1 year", "15 years");
   printf("-------------------------\n");

   return 0;
}
```

```
Dog age in human years (dogyears.com)

-------------------------
Dog age    | Human age
-------------------------
2 months   |    14 months
6 months   |     5 years
8 months   |      9 years
1 year     |     15 years
-------------------------
```

PARTICIPATION ACTIVITY 7.2.3: Formatting string outputs using printf().

Use myString below to determine the output from the print statements. Note: correct answers will be shown with quotations to denote string output and to show spacing.

```
char myString[30] = "Testing";
```


1) `printf("%4s", myString);`

Check Show answer

2) `printf("%8s", myString);`

Check Show answer

3) `printf("%.4s", myString);`

Check Show answer

4) `printf("%.10s", myString);`

Check Show answer

5) `printf("%-8s123", myString);`

Check Show answer

Flushing output

Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources; once those resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print. As such, the system may wait until the buffer is full, or at least has a certain number of characters before moving them to the output device. Or, with

fewer characters in the buffer, the system may wait until the resources are not busy. However, sometimes a programmer does not want the system to wait. For example, in a very processor-intensive program, such waiting could cause delayed and/or jittery output. The programmer can use the function ***fflush()***. The fflush() function will immediately flush the contents of the buffer for the specified FILE*. For example, fflush(stdout) will write the contents of the buffer for stdout to the computer screen.

Exploring further:

- More formatting options exist. See printf Reference Page from cplusplus.com.

CHALLENGE ACTIVITY 7.2.1: Output formatting.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>

int main(void) {
   int myInt;

   myInt = 341;

   printf("%1d\n", myInt);
   printf("%4d\n", myInt);

   return 0;
}
```

1 2 3 4 5

Check Next

CHALLENGE ACTIVITY 7.2.2: Output formatting.

Write a single statement that prints outsideTemperature with a + or - sign. End with newline. Sample output with input 103.5:

```
+103.500000
```

347282.2052980.qx3zqy7

```
#include <stdio.h>

int main(void) {
   double outsideTemperature;

   scanf("%lf", &outsideTemperature);

   /* Your solution goes here  */

   return 0;
}
```


Run

View your last submission

CHALLENGE ACTIVITY 7.2.3: Output formatting: Printing a maximum number of digits in the fraction.

Write a single statement that prints outsideTemperature with 2 digits in the fraction (after the decimal point). End with a newline. Sample output with input 103.45632:

```
103.46
```

347282.2052980.qx3zqy7

```
#include <stdio.h>

int main(void) {
   double outsideTemperature;

   scanf("%lf", &outsideTemperature);

   /* Your solution goes here  */

   return 0;
}
```

Run

View your last submission

7.3 Input parsing

This section describes features of the similar functions ***scanf***, ***fscanf***, and the soon-to-be-introduced sscanf, that support input parsing. The section illustrates using scanf, but the features apply to all three functions.

A programmer can control the way that input is read when using scanf(), a task known as ***input parsing***. The format specifiers within the format string of scanf() can include ***format sub-specifiers***. These sub-specifiers specify how the input will be read for that format specified. One of the most useful specifiers is the width specifier that can be used with the following form:

Construct 7.3.1: Format specifiers and sub-specifiers.

```
%(width)specifier
```

The width specifies the maximum number of character to read for the current format specifier. For example, the format string "%2d" will read in up to 2 characters -- in this case decimal digits -- converting the characters to the corresponding decimal value and storing that value into an integer variable.

A single scanf() statement can be used to read into multiple variables. The format string can include whitespace characters separating the format specifiers. These whitespace characters will cause the scanf() function to read all whitespace characters from the input until a non-whitespace character is reached. For example, the format string "%d %d" will read two decimal integers from the input separated by whitespace. That whitespace may be a single space, a newline, a space followed by a newline, or any combination thereof.

The following program uses a single scanf() statement to read two values for feet and inches, printing to equivalent distance in centimeters.

Figure 7.3.1: Reading multiple values using a single scanf().

```
#include <stdio.h>

const double CM_PER_IN = 2.54;
const int    IN_PER_FT = 12;

/* Converts a height in feet/inches to centimeters */
double HeightFtInToCm(int heightFt, int heightIn) {
   int totIn;
   double cmVal;

   totIn = (heightFt * IN_PER_FT) + heightIn; // Total inches
   cmVal = totIn * CM_PER_IN;                 // Conv inch to cm
   return cmVal;
}

int main(void) {
   int userFt;  // User defined feet
   int userIn;  // User defined inches

   // Prompt user for feet/inches
   printf("Enter feet and inches separated by a space: ");
   scanf("%d %d", &userFt, &userIn);

   // Output converted feet/inches to cm result
   printf("Centimeters: %lf\n",
          HeightFtInToCm(userFt,userIn));

   return 0;
}
```



```
Enter feet and inches separated by a space: 13 5
Centimeters: 408.940000

...

Enter feet and inches separated by a space: 3 5
Centimeters: 104.140000
```

PARTICIPATION ACTIVITY 7.3.1: Parsing input using scanf().

Answer the following questions assuming the user input is:

```
1053 17.5 42
```

1) What is the value of the variable val3 after the following scanf():

```
scanf("%d %f %d", &val1,
&val2, &val3);
```

Check Show answer

2) What is the value of the variable val3 after the scanf():

```
scanf("%2d %f %d", &val1,
&val2, &val3);
```


Check Show answer

User input often may include additional characters that are common to the format of the data being entered. For example, when receiving a time from a user, the programmer may prefer to allow users to use a common time format, such as "12:35 AM". In this example, the ':' is only used to format the data, separating the hour from the minute value.

The format string for scanf() can be configured to read the ':' character from the input but not store within a variable. scanf() will attempt to read any non-whitespace characters from the input. scanf() will only read the non-whitespace character if that character matches the provided user input.

Ex: the format string "%2d:%2d %2s" can be used to read in a time value:

- The first format specifier "%2d" will read up to two decimal digits for the hour.
- scanf() will then attempt to read a ':' character. If ':' is found in the user input, then ':' will be read and discarded.
- The subsequent two format specifiers will read in the minutes and AM/PM setting.

Figure 7.3.2: An example of using non-whitespace characters in a format string to parse formatted input.

```c
#include <stdio.h>
#include <string.h>

int main(void) {
   int currHour;      // User defined hour
   int currMinute;    // User defined minutes
   char optAmPm[3];   // User defined am/pm

   // Prompt user for input
   printf("Enter the time using the format: HH:MM AM/PM: ");
   scanf("%2d:%2d %2s", &currHour, &currMinute, optAmPm);

   // Output time in 12 hrs
   printf("In 12 hours it will be: ");
   if (strcmp(optAmPm, "AM") == 0) {
      printf("%02d:%02d PM\n", currHour, currMinute);
   }
   else {
      printf("%02d:%02d AM\n", currHour, currMinute);
   }

   return 0;
}
```



```
Enter the time using the format: HH:MM AM/PM: 12:35 PM
In 12 hours it will be: 12:35 AM

...

Enter the time using the format: HH:MM AM/PM: 4:12AM
In 12 hours it will be: 04:12 PM
```

Importantly, as soon as scanf() is not able to match the format string, it will stop reading from the input. For example, if the user does not enter the ':' character, scanf() will immediately stop reading from the input. In such a situation the currMinutes and optAmPm variables will not be updated.

zyDE 7.3.1: scanf() parsing.

Try running the program with the following user inputs

- 12:35 PM
- 12 35 PM
- "12 35 PM", "Time", "1235"

Load default template...

```c

#include <stdio.h>
#include <string.h>

int main(void) {
   int currHour;       // User defir
   int currMinute;     // User defir
   char optAmPm[3];    // User defir

   // Prompt user for input
```

12:35 PM

Run

```
11    printf("Enter the time using the
12    scanf("%2d:%2d %2s", &currHour, &
13
14    // Output time in 12 hrs
15    printf("In 12 hours it will be: "
16    if (strcmp(optAmPm, "AM") == 0) {
17       printf("%02d:%02d PM\n", currH
18    }
```


PARTICIPATION ACTIVITY 7.3.2: Parsing non-whitespace characters using scanf().

Assume all variables are initialized to zero. Answer the following questions assuming the user input is:

```
19, 20, 21
```

1) What is the value of the variable val2 after the scanf()?

```
scanf("%d %f %d", &val1,
&val2, &val3);
```

Check Show answer

2) What is the value of the variable val3 after the scanf()?

```
scanf("%d, %f, %d", &val1,
&val2, &val3);
```

Check Show answer

To check for such errors, the scanf() function returns an integer value for the number of items read using scanf() and stored within the specified variables. This return value can be checked to see if the user input matches the specified format. For example, if the user enters a valid time for the format string, scanf() will return 3. The following program extends the earlier example, printing an error message if the user input did not match the specified format string for all three format specifiers.

Figure 7.3.3: Using the return value from scanf() to check for parsing errors.

```c
#include <stdio.h>
#include <string.h>

int main(void) {
   int currHour;       // User defined hour
   int currMinute;     // User defined minutes
   char optAmPm[3];    // User defined am/pm

   // Prompt user for input
   printf("Enter the time using the format: HH:MM AM/PM: ");

   // Check number of items read
   if (scanf("%2d:%2d %2s", &currHour, &currMinute, optAmPm) != 3 ) {
      printf("\nInvalid time format\n");
   }
   else {
      printf("In 12 hours it will be: ");
      if (strcmp(optAmPm, "AM") == 0) {
         printf("%02d:%02d PM\n", currHour, currMinute);
      }
      else {
         printf("%02d:%02d AM\n", currHour, currMinute);
      }
   }
   return 0;
}
```



```
Enter the time using the format: HH:MM AM/PM: 12:35 PM
In 12 hours it will be: 12:35 AM

...

Enter the time using the format: HH:MM AM/PM: 412AM

Invalid time format
```


Sometimes a programmer wishes to read input data from a string rather than from the keyboard (standard input). The ***sscanf()*** function is used to read a sequence of characters from a C string, parsing the data stored within that string and storing the converted value within variables. The first

argument to sscanf() is the string being read. The remaining arguments for sscanf() work the same way as the arguments for scanf(). Specifically, the second argument for the sscanf() function is the ***format string*** that specifies the type of value to be read using a ***format specifier***. The argument following the format string is the location to store the values that are read.

Unlike the scanf() function that continues reading from the user input where the previous scanf() stopped, sscanf() always starts at the beginning of the specified string. In addition, the contents of the string being read are not modified by sscanf(). The following program illustrates.

Figure 7.3.4: Using sscanf() to parse a string.

```
#include <stdio.h>
#include <string.h>

int main(void) {
   char myString[100] = "Amy Smith 19";  // Input string
   char firstName[50];                   // First name
   char lastName[50];                    // Last name
   int userAge;                          // Age

   // Parse input, break up into first/last name and age
   sscanf(myString, "%49s %49s %d", firstName, lastName, &userAge);

   // Output parsed values
   printf("First name: %s\n", firstName);
   printf("Last name: %s\n", lastName);
   printf("Age: %d\n", userAge);

   return 0;
}
```

```
First name: Amy
Last name: Smith
Age: 19
```

A common use of scanf() is to process user input line-by-line. The following program reads in the line as a string, and then extracts individual data items from that string.

Figure 7.3.5: Using a sscanf() to parse a line of input text.

```c
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

int main(void) {
   const int USER_TEXT_LIMIT = 1000;    // Limit input size
   char userText[USER_TEXT_LIMIT];      // Holds input
   char firstName[50];                  // First name
   char lastName[50];                   // Last name
   int userAge;                         // Age
   int valuesRead;                      // Holds number of inputs read
   bool inputDone;                      // Flag to indicate next iteration

   inputDone = false;

   // Prompt user for input
   printf("Enter \"firstname lastname age\" on each line\n");
   printf("(\"Exit\" as firstname exits).\n\n");

   // Grab data as long as "Exit" is not entered
   while (!inputDone) {

      // Grab entire line, store in userText
      fgets(userText, USER_TEXT_LIMIT, stdin);

      // Parse the line and check for correct number of entries.
      valuesRead = sscanf(userText, "%49s %49s %d", firstName, lastName, &userAge);
      if (valuesRead >= 1 && strcmp(firstName, "Exit") == 0) {
         printf("Exiting.\n");
         inputDone = true;
      }
      else if (valuesRead == 3) {
         printf("   First name: %s\n", firstName);
         printf("   Last  name: %s\n", lastName);
         printf("   Age: %d\n", userAge);
         printf("\n");
      }
      else {
         printf("Invalid entry. Please try again.\n\n");
      }
   }

   return 0;
}
```



```
Enter "firstname lastname age" on each line
("Exit" as firstname exits).

Amy Smith 19
   First name: Amy
   Last  name: Smith
   Age: 19

Mike Smith 24
   First name: Mike
   Last  name: Smith
   Age: 24

No Age
Invalid entry. Please try again.

Exit
Exiting.
```

The program uses fgets() to read an input line into a string. Recall that C strings are implemented using character arrays. As the size of the character array -- or string -- must be known before calling fgets(), if the user enters a line of text that is longer than the length of that string, care must be taken to ensure the user input is not written to an out of bounds index.

The second argument to the fgets() function is an integer value specifying the maximum number of characters to write to the specified string. Using this input correctly ensures fgets() will not write to out of range values for the specified string. For example, if inputBuffer is declared as `char inputBuffer[100]`, the statement `fgets(inputBuffer, 100, stdin);` will ensure that no more than 100 characters are written to the string inputBuffer. Additionally, fgets() will ensure that the null character will be written to the end of the string read.

Similarly, when parsing a string -- or user input -- to read a string, the width sub-specifier of the "%s" format specifier should be used. Recall that the width sub-specifier specifies the maximum number of characters to read. If myString is defined char myString[50], the format specifier "%49s" can be used to ensure no more than 49 characters are read from the input, leaving one space for the null character at the end of the string.

A good practice is to always use the width sub-specifier when reading strings using scanf(), fscanf(), or sscanf().

PARTICIPATION ACTIVITY 7.3.3: More input parsing.

Answer the following questions assuming the user input is:

```
1053 17.5 42 Smith
```

1) What is the value of the variable str2 after the scanf() (include quotes in your answer)?

```
scanf("%s %d %s", str1,
&val1, str2);
```

Check Show answer

2) What is the return value from the following scanf():

```
scanf("%f %d %d %d", &val1,
&val2, &val3, &val4);
```

Check Show answer

3) What is the value of the variable str3 after the fgets() (include quotes in your answer)? Assume USER_TEXT_LIMIT is 1000.

```
fgets(str3,
USER_TEXT_LIMIT, stdin);
```

Check Show answer

Exploring further:

- getc() from cplusplus.com
- getchar() from cplusplus.com

CHALLENGE ACTIVITY 7.3.1: Input parsing.

347282.2052980.qx3zqy7

Start

Type the program's output

```c
#include <stdio.h>
#include <string.h>

int main(void) {
   char objectInfo[100] = "Shoes 16 17";
   char object[50];
   int quantity;
   int price;

   sscanf(objectInfo, "%s %d %d", object, &price, &quantity);

   printf("%s x%d\n", object, quantity);
   printf("Price: %d", price);

   return 0;
}
```


1 2

Check Next

CHALLENGE ACTIVITY 7.3.2: Input parsing: Reading an entire line.

Write a single statement that reads an entire line from stdin. Assign streetAddress with the user input. Ex: If a user enters "1313 Mockingbird Lane", program outputs:

```
You entered: 1313 Mockingbird Lane
```

347282.2052980.qx3zqy7

```c
#include <stdio.h>

int main(void) {
   const int ADDRESS_SIZE_LIMIT = 50;
   char streetAddress[ADDRESS_SIZE_LIMIT];

   printf("Enter street address: ");

   /* Your solution goes here  */

   printf("You entered: %s", streetAddress);

   return 0;
}
```


Run

View your last submission

CHALLENGE ACTIVITY 7.3.3: Input parsing: Reading multiple items.

Complete scanf() to read two comma-separated integers from stdin. Assign userInt1 and userInt2 with the user input. Ex: "Enter two integers separated by a comma: 3, 5", program outputs:

```
3 + 5 = 8
```

347282.2052980.qx3zqy7

```
1  #include <stdio.h>
2
3  int main(void) {
4     int userInt1;
5     int userInt2;
6
7     printf("Enter two integers separated by a comma: ");
8     scanf(/* Your solution goes here  */);
9     printf("%d + %d = %d\n", userInt1, userInt2, userInt1 + userInt2);
10
11    return 0;
12 }
```

Run

View your last submission

7.4 File input and output

Sometimes a program should get input from a file rather than from a user typing on a keyboard. To achieve this, a programmer can open another input file, rather than the predefined input file stdin that

comes from the standard input (keyboard). That new input file can then be used with fscanf() just like using scanf() with the stdin file, as the following program illustrates. Assume a text file exists named myfile.txt with the contents shown (created for example using Notepad on a Windows computer or using TextEdit on a Mac computer).

Figure 7.4.1: Input from a file.


```c
#include <stdio.h>

int main(void) {
   FILE* inFile = NULL; // File pointer
   int fileNum1;        // Data value from file
   int fileNum2;        // Data value from file

   // Try to open file
   printf("Opening file myfile.txt.\n");

   inFile = fopen("myfile.txt", "r");
   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Can now use fscanf(inFile, ...) like scanf()
   // myfile.txt should contain two integers, else problems
   printf("Reading two integers.\n");
   fscanf(inFile, "%d %d", &fileNum1, &fileNum2);

   // Done with file, so close it
   printf("Closing file myfile.txt.\n");
   fclose(inFile);

   // Output values read from file
   printf("num1 = %d\n", fileNum1);
   printf("num2 = %d\n", fileNum2);
   printf("num1+num2 = %d\n", (fileNum1 + fileNum2));

   return 0;
}
```

myfile.txt with two integers:

```
5
10
```

```
Opening file myfile.txt.
Reading two integers.
Closing file myfile.txt.
num1 = 5
num2 = 10
num1+num2 = 15
```

Six lines are needed for input from a file, highlighted above.

- The `#include <stdio.h>` enables use of FILE* variables and supporting functions.
- A new FILE* variable has been declared: `FILE* inputFile;`. FilePointer
- The line `inputFile = fopen("myfile.txt", "r");` then opens the file for reading and associates the file with the FILE*. The first argument to ***fopen()*** is a string with the name of the file to open. The second argument of fopen() is a string indicating the file mode, which specifies if the file should be open for reading or writing. The string "r" indicates the file should be open for reading, referred to as ***read mode***. Upon success, fopen() will return a pointer to the FILE structure for the file that was opened. If fopen() could not open the file, it will return NULL.

- Because of the high likelihood that the open fails, usually because the file does not exist or is in use by another program, the program checks whether the open was successful using `if (inputFile == NULL)`.
- The successfully opened input file is read from using fscanf(), e.g., using `fscanf(inFile, "%d %d", &num1, &num2);` to read two integers into num1 and num2.
- Finally, when done using the file, the program closes the file using `fclose(inputFile);`.

A common error is to specify the file mode as a character (e.g. 'r') rather than a string (e.g."r"). Another common error is a mismatch between the variable data type and the file data, e.g., if the data type is int but the file data is "Hello".

Try 7.4.1: Good and bad file data.

File input, with good and bad data: Create myfile.txt with contents 5 and 10, and run the above program. Then, change "10" to "Hello" and run again, observing the incorrect output.

The following provides another example wherein the program reads items into a dynamically allocated array. For this program, myfile.txt's first entry must be the number of numbers to read, followed by those numbers, e.g., 5 10 20 40 80 1.

Figure 7.4.2: Program that reads data from myfile.txt into an array.

myfile.txt file contents:

```
5
10
20
40
80
1
```

```
Numbers: 10 20 40 80 1
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   FILE* inFile = NULL; // File pointer
   int* userNums;       // User numbers; memory allocated later
   int arrSize;         // User-specified number of numbers
   int i;               // Loop index


   // Try to open the file
   inFile = fopen("myfile.txt", "r");

   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Can now use fscanf(inFile, ...) like scanf()
   fscanf(inFile, "%d", &arrSize);

   // Allocate enough memory for nums
   userNums = (int*)malloc(sizeof(int)*arrSize);
   if (userNums == NULL) {
      fclose(inFile); // Done with file, so close it
      return -1;
   }

   // Get user specified numbers. If too few, may encounter problems
   i = 1;
   while (i <= arrSize) {
      fscanf(inFile, "%d", &(userNums[i-1]));
      i = i + 1;
   }

   // Done with file, so close it
   fclose(inFile);

   // Print numbers
   printf("Numbers: ");

   i = 0;
   while (i < arrSize) {
      printf("%d ", userNums[i]);
      ++i;
   }

   printf("\n");

   // Deallocate memory for nums
   free(userNums);

   return 0;
}
```


©zyBooks 12/22/21 12:33 1026490
Jacob Adams
UACS100Fall2021

A program can read varying amounts of data in a file by using a loop that reads until the end of the file has been reached, as follows.

The ***feof()*** function returns 1 if the previous read operation reached the end of the file. Errors may be encountered while attempting to read from a file, including end-of-file, corrupt data, etc. So, a program should check that each read was successful before using the variable to which the data read was assigned. fscanf() returns the number of items read from the file and assigned to a variable, which can be checked to determine if the read operation was successful. Ex:
`if( fscanf(inFile, "%d", &fileNum) == 1 ) {...}` checks that fscanf() read and assigned a value to fileNum.

Figure 7.4.3: Reading a varying amount of data from a file.

```
#include <stdio.h>

int main(void) {
   FILE* inFile = NULL; // File pointer
   int fileNum;         // Data value from file
   int numRead;

   // Open file
   printf("Opening file myfile.txt.\n");
   inFile = fopen("myfile.txt", "r");

   if (inFile == NULL) {
      printf("Could not open file myfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Print read numbers to output
   printf("Reading and printing numbers.\n");

   numRead = fscanf(inFile, "%d", &fileNum);

   while (!feof(inFile)) {
      if ( numRead == 1 ) {
         printf("num: %d\n", fileNum);
      }
      else {
         printf("Input failure before reaching end of
file.\n");
         break;
      }
      numRead = fscanf(inFile, "%d", &fileNum);
   }

   printf("Closing file myfile.txt.\n");

   // Done with file, so close it
   fclose(inFile);

   return 0;
}
```

myfile.txt with variable number of integers:

```
111
222
333
444
555
```

```
Opening file myfile.txt.
Reading and printing numbers.
num: 111
num: 222
num: 333
num: 444
num: 555
Closing file myfile.txt.
```


Similarly, a program may write output to a file rather than to standard output, as shown below. To open an output file, the string "w" is used as the file mode within the call to fopen(), referred to as ***write***

mode. Using the write mode, if a file with specified name already exists, that file will be replaced with the newly created file.

Figure 7.4.4: Sample code for writing to a file.

```
#include <stdio.h>

int main(void) {
   FILE* outFile = NULL;    // File pointer

   // Open file
   outFile = fopen("myoutfile.txt", "w");

   if (outFile == NULL) {
      printf("Could not open file
myoutfile.txt.\n");
      return -1; // -1 indicates error
   }

   // Write to file
   fprintf(outFile, "Hello\n");
   fprintf(outFile, "1 2 3\n");

   // Done with file, so close it
   fclose(outFile);

   return 0;
}
```


Contents of myoutfile.txt after running the program:

```
Hello
1 2 3
```

fopen() supports several additional file modes. See http://www.cplusplus.com/reference/cstdio/fopen/.

PARTICIPATION ACTIVITY 7.4.1: Opening file using open().

Answer the following assuming the file "file1.txt" exists and can be accessed by the user and "file2.txt" does not exist.

1) Write a statement to open the "file1.txt" for input, assigning the return from fopen() to a FILE* variable named inputFile.

Check Show answer

2) What is the value of the FILE*

inputFile after the following call to fopen(): `inputFile = fopen("file2.txt", "r");`

Check Show answer

3) Write a statement to open the "file2.txt" for output, assigning the return from fopen() to a FILE* variable named outputFile.

Check Show answer

4) Write a statement that can read in data from an already established input file `inputFile` until the end of file has been reached.

```
while (                    )
{
   // Read/manipulate file
data
}
```

Check Show answer

Exploring further:

- stdlib.h reference page from cplusplus.com

(*FilePointer) Pointers are described in another section. Knowledge of that section is not essential to understanding the current section.

7.5 Command-line arguments

Command-line arguments are values entered by a user when running a program from a command line. A *command line* exists in some program execution environments, wherein a user types a

program's name and any arguments at a command prompt. To access those arguments, main() can be defined with two special parameters argc and argv, as shown below. The program prints provided command-line arguments. (The "for" loop is not critical to understanding the point, in case you haven't studied for loops yet). The program's executable is named argtest.

Figure 7.5.1: Printing command-line arguments.


```
#include "stdio.h"

int main(int argc, char* argv[]) {
   int i;

   // Prints argc and argv values
   printf("argc: %d\n", argc);
   for (i = 0; i < argc; ++i) {
      printf("argv[%d]: %s\n", i, argv[i]);
   }

   return 0;
}
```

```
> ./argtest
argc: 1
argv[0]: ./argtest

> ./argtest Hello
argc: 2
argv[0]: ./argtest
argv[1]: Hello

> ./argtest Hey ABC 99 -5
argc: 5
argv[0]: ./argtest
argv[1]: Hey
argv[2]: ABC
argv[3]: 99
argv[4]: -5
```

When a program is run, the system passes an int parameter ***argc*** to main(), indicating the number of command-line arguments (argc is short for argument count). The number includes the program name itself, so argc is 2 for the command line: `myprog.exe myfile.txt`.

When a program is run, the system passes a second parameter ***argv*** to main() (argv is short for argument vector), defined as an array of strings: char* argv[]. argv[] consists of one string for each command-line argument, with argv[0] being the program name.

PARTICIPATION ACTIVITY 7.5.1: Command-line arguments.

Animation captions:

1. The system passes an int parameter argc to main(), which indicates the number of command-line arguments.
2. The system passes a second parameter argv to main(), which is an array of strings consisting of one string for each command-line argument.

PARTICIPATION ACTIVITY 7.5.2: Command-line arguments.

1) What is argc for: :

```
myprog.exe 13 14 smith
```

Check Show answer

2) What is argc for:

```
a.out 12:55 PM
```


Check Show answer

3) What is the string in argv[2] for:

```
a.out Jan Feb Mar
```

Check Show answer

The following program, named myprog, expects two command-line arguments.

Figure 7.5.2: Simple use of command-line arguments.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   char nameStr[100];       // User name
   char ageStr[100];        // User age

   // Get inputs from command line
   strcpy(nameStr, argv[1]);
   strcpy(ageStr, argv[2]);

   // Output result
   printf("Hello %s. ", nameStr);
   printf("%s is a great age.\n", ageStr);

   return 0;
}
```

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Rajeev 44 HEY
Hello Rajeev. 44 is a great age.

...

> myprog.exe Denming
Segmentation fault
```


For simplicity, the above program used character arrays for the string variables name and ageStr. For those familiar with char* and malloc(), dynamically allocating the appropriately-sized C strings would be preferred.

However, there is no guarantee a user will type two command-line arguments. Extra arguments, like "HEY" above, are ignored. Conversely, too few arguments can cause a problem. In particular, a common error is to access elements in argv without first checking argc to ensure the user entered enough arguments, resulting in an out-of-range array access. In the last run above, the user typed too few arguments, causing an out-of-range array access.

When a program uses command-line arguments, good practice is to check argc for the correct number of arguments. If the number of command-line arguments is incorrect, good practice is to print a usage message. A ***usage message*** lists a program's expected command-line arguments. The program should then return 1, indicating to the system that an error occurred (whereas 0 means no error).

Figure 7.5.3: Checking for proper number of command-line arguments.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Usage: program username userage */
int main(int argc, char* argv[]) {
   char nameStr[100];       // User name
   char ageStr[100];        // User age

   // Check if correct number of arguments provided
   if (argc != 3) {
      printf("Usage: myprog.exe name age\n");
      return 1; // 1 indicates error
   }

   // Grab inputs from command line
   strcpy(nameStr, argv[1]);
   strcpy(ageStr, argv[2]);

   // Output result
   printf("Hello %s. ", nameStr);
   printf("%s is a great age.\n", ageStr);

   return 0;
}
```

```
> myprog.exe Amy 12
Hello Amy. 12 is a great age.

...

> myprog.exe Denming
Usage: myprog.exe name age

...

> myprog.exe Alex 26 pizza
Usage: myprog.exe name age
```

PARTICIPATION ACTIVITY 7.5.3: Checking the number of command-line arguments.

1) If a user types the wrong number of command-line arguments, good practice is to print a usage message.
   - True
   - False

2) If a user types too many arguments but

a program doesn't check for that, the program typically crashes.

- True
- False

3) If a user types too few arguments but a program doesn't check for that, the program typically crashes.

- True
- False

Command-line arguments are C strings. The statement `age = atoi(ageStr);` converts the ageStr string into an integer, assigning the result into int variable age. So string "12" becomes integer 12. atoi() converts a C string to an integer, and is made available via `#include <stdlib.h>`.

Putting quotes around an argument allows an argument's string to have any number of spaces.

Figure 7.5.4: Quotes surround the single argument 'Mary Jo'.

```
myprog.exe "Mary Jo" 50
```

PARTICIPATION ACTIVITY 7.5.4: String and integer command-line arguments.

1) What is the string in argv[1] for the following:

```
a.out Amy Smith 19
```

Check Show answer

2) What is the string in argv[1] for the following:

```
a.out "Amy Smith" 19
```

Check Show answer

3) Given the following code snippet, complete the assignment of userNum with argv[1].

```
int main(int argc, char* argv[]) {
   int userNum;
   userNum = /* COMPLETE ASSIGNMENT
*/;
}
```

userNum = ;

Check Show answer

Exploring further:

- Command-line arguments from cplusplus.com (Applies equally to C)

7.6 Command-line arguments and files

The location of an input file or output file may not be known before writing a program. Instead, a program can use command-line arguments to allow the user to specify the location of an input or output file as shown in the following program. Assume two text files exist in the same directory as the source code named "myfile1.txt" and "myfile2.txt" with the contents shown. The sample output shows the results when executing the program for each input file and for an input file that does not exist.

Figure 7.6.1: Using command-line arguments to specify the name of an input file.

myfile1.txt:

```
5
10
```


myfile2.txt:

```
-34
7
```

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
  FILE* inFile = NULL; // File pointer
  int fileNum1;        // Data value from file
  int fileNum2;       // Data value from file

  // Check number of arguments
  if( argc != 2 ) {
    printf("Usage: myprog.exe inputFileName\n");\
    return 1; // 1 indicates error
  }

  // Try to open the file
  printf("Opening file %s.\n", argv[1]);

  inFile = fopen(argv[1], "r");
  if( inFile == NULL ) {
    printf("Could not open file %s.\n", argv[1]);
    return -1; // -1 indicates error
  }

  // Can now use fscanf(inFile, ...) like scanf()
  // myfile.txt should contain two integers, else problems
  printf("Reading two integers.\n");
  fscanf(inFile, "%d %d", &fileNum1, &fileNum2);

  // Done with file, so close it
  printf("Closing file %s.\n\n", argv[1]);
  fclose(inFile);

  // Output values read from file
  printf("num1: %d\n", fileNum1);
  printf("num2: %d\n", fileNum2);
  printf("num1 + num2: %d\n", (fileNum1 + fileNum2));

  return 0;
}
```

```
> myprog.exe myfile1.txt
Opening file myfile1.txt.
Reading two integers.
Closing file myfile1.txt.

num1: 5
num2: 10
num1 + num2: 15

...
> myprog.exe myfile2.txt
Opening file myfile2.txt.
Reading two integers.
Closing file myfile2.txt.

num1: -34
num2: 7
num1 + num2: -27

...

> myprog.exe myfile3.txt
Opening file myfile3.txt.
Could not open file
myfile3.txt.
```


PARTICIPATION ACTIVITY 7.6.1: Filename command-line arguments.

1) A program "myprog.exe" has two command-line arguments, one for an input file and a second for an output file. Type a command to run the program with input file "infile.txt" and output file "out".

Check Show answer

2) For a program run as `progname data.txt`, what is argv[1]? Don't use quotes in your answer.

Check Show answer

7.7 Binary file I/O

This section has been set as optional by your instructor.

Programs can also access files using a ***binary file mode***. Using the binary file mode, a program can directly copy data in the program's memory to and from the file. This mode is referred to as binary mode because reading and writing to the file will copy the contents to and from memory bit by bit.

Binary files provide several advantages.

- Reading from and writing to binary files is very efficient because these operations directly copy data without first converting the data into a human readable format. For example, consider writing a 32-bit integer value, such as -13645766, to an output file. Using a character file, this 32-bit integer would first need to be converted to the sequence of characters "-13645766" representing that integer value. When writing to a binary file, this conversion is not needed.
- Binary files can be more space efficient than character files. Again, consider writing the same integer value -13645766 to an output file. Using a binary file, the value requires 32-bits -- or 4 bytes -- as the integer value is stored in memory using 4 bytes. Using a character file would require 9 bytes -- one byte for each character within "-13645766".

Binary files also have disadvantages. The main disadvantage of binary files is that they cannot be easily read or edited by humans. A good practice is to only use binary files when those files will never be directly read or edited by a user.

PARTICIPATION ACTIVITY 7.7.1: Binary files basics.

1) Binary files are human readable.

- True
- False

2) Binary files are both more efficient with memory space, as well as, reading and writing operations.

- True
- False

To open a file using a binary mode, the file mode string in the call to fopen() should include the character b. To open a file for reading using the binary mode, the file mode string "rb" would be used. Similarly, to open a file for writing using the binary mode, the file mode string "wb" would be used.

A program can write to a binary file using the fwrite() function using the following form:

Construct 7.7.1: fwrite() function.

```
fwrite(variablePointer, sizeof(type), numElements, outputFile)
```

The first argument of fwrite() is a pointer to the variable, or memory location, that will be copied and written to the binary file. The second and third arguments specify the size in bytes of each element and the number of elements, respectively, being written to the file. The sizeof() operation can be used to determine the number of bytes required for the data type of the variable or array being written. The final argument to fwrite() is a FILE* for the output file being written. fwrite() will return the total number of bytes written to the output file.

The following demonstrates the use of fwrite() to write several user-entered integer numbers to a binary file.

Figure 7.7.1: Sample code for writing an array of integers one-by-one to a binary file.

```
Enter 8 numbers...
0: 567
1: 342
2: 7
3: 1000000
4: 34965
5: 42
6: 1700
7: -25
Writing numbers to myfile.bin.
Closing file myfile.bin.
```

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   const int NUM_VALUES = 8; // Num user numbers
   FILE* outFile = NULL;     // File pointer
   int userNums[NUM_VALUES]; // User numbers
   int i;                    // Loop index

   // Get numbers from user
   printf("\nEnter %d numbers...\n", NUM_VALUES);
   for (i = 0; i < NUM_VALUES; ++i) {
      printf("%d: ", i);
      scanf("%d", &(userNums[i]));
   }

   // Try to open the file
   outFile = fopen("myfile.bin", "wb");
   if( outFile == NULL ) {
      printf("Could not open file myfile.bin.\n");
      return -1; // -1 indicates error
   }

   // Write user values to output file
   printf("Writing numbers to myfile.bin.\n");
   for (i = 0; i < NUM_VALUES; ++i) {
      fwrite(&(userNums[i]), sizeof(int), 1, outFile);
   }

   // Done with file, so close it
   printf("Closing file myfile.bin.\n");
   fclose(outFile);

   return 0;
}
```


Try 7.7.1: Viewing a binary file in a text editor.

Run the above program and try opening the resulting myfile.bin file using a text editor. Notice that the contents of the file are not easily readable.

As the elements of an array are stored in sequential memory locations, the fwrite() command can also be used to directly copy the entire contents of an array to the output file by passing the number of elements within the array as third argument to fwrite(). The following programs illustrates.

Figure 7.7.2: Sample code for writing an array of integers to a binary file using a single fwrite() call.

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   const int NUM_VALUES = 8; // Num user numbers
   FILE* outFile = NULL;     // File pointer
   int userNums[NUM_VALUES]; // User numbers
   int i;                    // Loop index


   // Get numbers from user
   printf("\nEnter %d numbers...\n", NUM_VALUES);
   for (i = 0; i < NUM_VALUES; ++i) {
      printf("%d: ", i);
      scanf("%d", &(userNums[i]));
   }

   // Try to open the file
   outFile = fopen("myfile.bin", "wb");
   if( outFile == NULL ) {
      printf("Could not open file myfile.bin.\n");
      return -1; // -1 indicates error
   }

   // Write entire int array on N elements to output file
   fwrite(userNums, sizeof(int), NUM_VALUES, outFile);

   // Done with file, so close it
   printf("Closing file myfile.bin.\n");
   fclose(outFile);

   return 0;
}
```



```
Enter 8 numbers...
0: 567
1: 342
2: 7
3: 1000000
4: 34965
5: 42
6: 1700
7: -25
Writing numbers to myfile.bin.
Closing file myfile.bin.
```

A program can read from a binary file using the fread() function using the following form:

Construct 7.7.2: fread() function.

```c
fread(variablePointer, sizeof(type), numElements, inputFile)
```


The first argument of fread() is a pointer to the variable in which the contents read from the input file will be copied. The second and third arguments specify the size in bytes of each element and the number of elements, respectively, being read from the file. The final argument to fread() is a FILE* for the input file being read. fread() will return the total number of bytes successfully read from the file.

The following programs uses fread() to read and print integer values from a binary input file until the end of the file is reached.

Figure 7.7.3: Sample code for reading and printing integers values read from a binary file using fread().

```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
   FILE* inFile = NULL;    // File pointer
   int fileNum;            // Data value from file
   long numBytesRead;      // Num bytes read from file

   // Try to open the file
   printf("Opening file myfile.bin.\n");

   inFile = fopen("myfile.bin", "rb");
   if( inFile == NULL ) {
      printf("Could not open file myfile.bin.\n");
      return -1; // -1 indicates error
   }

   // Print read numbers to output
   printf("Reading and printing numbers.\n");
   numBytesRead = fread(&fileNum, sizeof(int), 1, inFile);

   while (numBytesRead != 0) {
      printf("num: %d\n", fileNum);
      numBytesRead = fread(&fileNum, sizeof(int), 1, inFile);
   }

   // Done with file, so close it
   printf("Closing file myfile.bin.\n");
   fclose(inFile);

   return 0;
}
```



```
Opening file myfile.bin.
Reading and printing numbers.
num: 567
num: 342
num: 7
num: 1000000
num: 34965
num: 42
num: 1700
num: -25
Closing file myfile.bin.
```

PARTICIPATION ACTIVITY 7.7.2: Opening file using open().

1) Write a statement to open the "file1.bin" for reading using binary mode and a FILE* variable named inputFile.

Check Show answer

2) Assuming an int is 32 bits and nums is an array of int, how many bytes will be written to the output file for the following call to fwrite():

```
fwrite(nums, sizeof(int),
7, outputFile);
```

Check Show answer

3) Write a single fread() statement to read three float values into an array named myVals using a FILE* variable named inputFile.

Check Show answer

Exploring further:

- stdio.h Reference Page from cplusplus.com
- fread() Reference Page from cplusplus.com
- fwrite() Reference Page from cplusplus.com
- C File IO Tutorial from cprogramming.com

7.8 LAB: Warm up: Parsing strings

(1) Prompt the user for a string that contains two strings separated by a comma. (1 pt)

- Examples of strings that can be accepted:
  - Jill, Allen
  - Jill , Allen
  - Jill,Allen

Ex:

```
Enter input string:
Jill, Allen
```

(2) Report an error if the input string does not contain a comma. Continue to prompt until a valid

string is entered. *Note: If the input contains a comma, then assume that the input also contains two strings.* (2 pts)

Ex:

```
Enter input string:
Jill Allen
Error: No comma in string.

Enter input string:
Jill, Allen
```


(3) Extract the two words from the input string and remove any spaces. Store the strings in two separate variables and output the strings. (2 pts)

Ex:

```
Enter input string:
Jill, Allen
First word: Jill
Second word: Allen
```

(4) Using a loop, extend the program to handle multiple lines of input. Continue until the user enters q to quit. (2 pts)

Ex:

```
Enter input string:
Jill, Allen
First word: Jill
Second word: Allen

Enter input string:
Golden , Monkey
First word: Golden
Second word: Monkey

Enter input string:
Washington,DC
First word: Washington
Second word: DC
```

```
Enter input string:
q
```

NaN.2052980.qx3zqy7

LAB ACTIVITY 7.8.1: LAB: Warm up: Parsing strings 0 / 7

main.c Load default template...

```
1 #include<stdio.h>
2 #include <string.h>
3
4 int main(void) {
5
6    /* Type your code here. */
7
8    return 0;
9 }
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

7.9 LAB: Thesaurus

Given a set of text files containing synonyms for different words, complete the main program to output the synonyms for a specific word. Each text file contains synonyms for the word specified in the file's name, and the synonyms within the file are grouped according to the synonyms' first letters, separated by an '*'. The program reads a word and a letter from the user and opens the text file associated with the input word. The program then stores the contents of the text file into a two-dimensional array of char* predefined in the program. Finally the program searches the array and outputs all the synonyms that begin with the input letter, one synonym per line, or a message if no synonyms that begin with the input letter are found.

Hints: Use the malloc() function to allocate memory for each of the synonyms stored in the array. A string always ends with a null character ('\0'). Use ASCII values to map the row index of the array to the first letter of a word when storing the synonyms into the array. Ex: Index 0 to an 'a', index 25 to a 'z'. Assume all letters are in lowercase.

Ex: If the input of the program is:

```
educate c
```

the program opens the file educate.txt, which contains:

```
brainwash
brief
*
civilize
coach
cultivate
*
develop
discipline
drill
*
edify
enlighten
exercise
explain
*
foster
*
```

```
improve
indoctrinate
inform
instruct
*
mature
*
nurture
*
rear
*
school
*
train
tutor
*
```


then the program outputs:

```
civilize
coach
cultivate
```

Ex: If the input of the program is:

```
educate a
```

then the program outputs:

```
No synonyms for educate begin with a.
```

NaN.2052980.qx3zqy7

LAB ACTIVITY | 7.9.1: LAB: Thesaurus | 0 / 10

Downloadable files

`educate.txt` **Download**

main.c Load default template...

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
```

```
int main(void) {
   int NUM_CHARACTERS = 26;   // Maximum number of letters
   int MAX_SYNONYMS = 10;     // Maximum number of synonyms per starting letter
   int MAX_WORD_SIZE = 30;    // Maximum length of the input word
   int n;
   char* synonyms[NUM_CHARACTERS][MAX_SYNONYMS];   // Declare 2D array of string p

   // Initialize the first column of the 2D array
   for (n = 0; n < NUM_CHARACTERS; n++) {
      synonyms[n][0] = NULL;
   }

   /* Type your code here. */

```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

7.10 LAB*: Program: Data visualization

This section has been set as optional by your instructor.

(1) Prompt the user for a title for data. Output the title. (1 pt)

Ex:

```
Enter a title for the data:
Number of Novels Authored
You entered: Number of Novels Authored
```


(2) Prompt the user for the headers of two columns of a table. Output the column headers. (1 pt)

Ex:

```
Enter the column 1 header:
Author name
You entered: Author name

Enter the column 2 header:
Number of novels
You entered: Number of novels
```

(3) Prompt the user for data points. Data points must be in this format: *string, int*. Store the information before the comma into a string variable and the information after the comma into an integer. The user will enter **`-1`** when they have finished entering data points. Output the data points. Store the string components of the data points in an array of strings. Store the integer components of the data points in an array of integers. (4 pts)

Ex:

```
Enter a data point (-1 to stop input):
Jane Austen, 6
Data string: Jane Austen
Data integer: 6
```


(4) Perform error checking for the data point entries. If any of the following errors occurs, output the appropriate error message and prompt again for a valid data point.

- If entry has no comma
  - Output: **`Error: No comma in string.`** (1 pt)
- If entry has more than one comma
  - Output: **`Error: Too many commas in input.`** (1 pt)
- If entry after the comma is not an integer

- Output: `Error: Comma not followed by an integer.` (2 pts)

Ex:

```
Enter a data point (-1 to stop input):
Ernest Hemingway 9
Error: No comma in string.

Enter a data point (-1 to stop input):
Ernest, Hemingway, 9
Error: Too many commas in input.

Enter a data point (-1 to stop input):
Ernest Hemingway, nine
Error: Comma not followed by an integer.

Enter a data point (-1 to stop input):
Ernest Hemingway, 9
Data string: Ernest Hemingway
Data integer: 9
```


(5) Output the information in a formatted table. The title is right justified with a width of 33. Column 1 has a width of 20. Column 2 has a width of 23. (3 pts)

Ex:

```
        Number of Novels Authored
Author name         |       Number of novels
-----------------------------------------
Jane Austen         |                    6
Charles Dickens     |                   20
Ernest Hemingway    |                    9
Jack Kerouac        |                   22
F. Scott Fitzgerald |                    8
Mary Shelley        |                    7
Charlotte Bronte    |                    5
Mark Twain          |                   11
Agatha Christie     |                   73
Ian Flemming        |                   14
Stephen King        |                   54
Oscar Wilde         |                    1
```

(6) Output the information as a formatted histogram. Each name is right justified with a width of 20. (4 pts)

Ex:

```
         Jane Austen ******
      Charles Dickens ******************
     Ernest Hemingway *********
         Jack Kerouac **********************
 F. Scott Fitzgerald ********
         Mary Shelley *******
     Charlotte Bronte *****
           Mark Twain ***********
      Agatha Christie
**************************************************************************************

         Ian Flemming **************
         Stephen King
******************************************************
          Oscar Wilde *
```


LAB ACTIVITY 7.10.1: LAB*: Program: Data visualization 0 / 17

main.c Load default template...

```
1 #include<stdio.h>
2 #include <string.h>
3
4 int main(void) {
5
6    /* Type your code here. */
7
8    return 0;
9 }
```

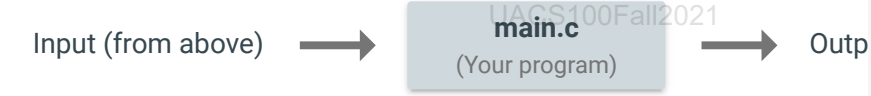
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

7.11 LAB: Parsing dates

This section has been set as optional by your instructor.

Complete main() to read dates from input, one date per line. Each date's format must be as follows: March 1, 1990. Any date not following that format is incorrect and should be ignored. Use the substring() method to parse the string and extract the date. The input ends with -1 on a line alone. Output each correct date as: 3/1/1990.

Ex: If the input is:


```
March 1, 1990
April 2 1995
7/15/20
December 13, 2003
-1
```

then the output is:

```
3/1/1990
12/13/2003
```


LAB ACTIVITY | 7.11.1: LAB: Parsing dates

0 / 10

main.c Load default template...

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int GetMonthAsInt(char *monthString) {
5     int monthInt;
6
7     if (strcmp(monthString, "January") == 0) {
8        monthInt = 1;
9     }
10    else if (strcmp(monthString, "February") == 0) {
11       monthInt = 2;
12    }
13    else if (strcmp(monthString, "March") == 0) {
14       monthInt = 3;
15    }
16    else if (strcmp(monthString, "April") == 0) {
17       monthInt = 4;
18    }
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


7.12 LAB: File name change

This section has been set as optional by your instructor.

A photographer is organizing a photo collection about the national parks in the US and would like to annotate the information about each of the photos into a separate set of files. Write a program that reads the name of a text file containing a list of photo file names. The program then reads the photo file names from the text file, replaces the "_photo.jpg" portion of the file names with "_info.txt", and outputs the modified file names.

Assume the unchanged portion of the photo file names contains only letters and numbers, and the text file stores one photo file name per line. If the text file is empty, the program produces no output. Assume also the maximum number of characters of all file names is 100.

Ex: If the input of the program is:

```
ParkPhotos.txt
```

and the contents of ParkPhotos.txt are:

```
Acadia2003_photo.jpg
AmericanSamoa1989_photo.jpg
BlackCanyonoftheGunnison1983_photo.jpg
CarlsbadCaverns2010_photo.jpg
CraterLake1996_photo.jpg
GrandCanyon1996_photo.jpg
IndianaDunes1987_photo.jpg
LakeClark2009_photo.jpg
Redwood1980_photo.jpg
VirginIslands2007_photo.jpg
Voyageurs2006_photo.jpg
WrangellStElias1987_photo.jpg
```


the output of the program is:

```
Acadia2003_info.txt
AmericanSamoa1989_info.txt
BlackCanyonoftheGunnison1983_info.txt
CarlsbadCaverns2010_info.txt
CraterLake1996_info.txt
GrandCanyon1996_info.txt
IndianaDunes1987_info.txt
LakeClark2009_info.txt
Redwood1980_info.txt
VirginIslands2007_info.txt
Voyageurs2006_info.txt
WrangellStElias1987_info.txt
```


LAB ACTIVITY 7.12.1: LAB: File name change 0 / 10

Downloadable files

`ParkPhotos.txt` **Download**

main.c Load default template...

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(void) {
5
6     /* Type your code here. */
7
8     return 0;
9  }
10
```

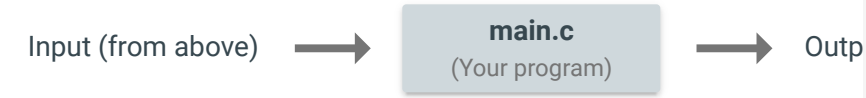

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

7.13 LAB: Course Grade

This section has been set as optional by your instructor.

Write a program that reads the student information from a tab separated values (tsv) file. The program then creates a text file that records the course grades of the students. Each row of the tsv file contains the Last Name, First Name, Midterm1 score, Midterm2 score, and the Final score of a student. A sample of the student information is provided in StudentInfo.tsv. Assume the number of students is at least 1 and at most 20.

The program performs the following tasks:

- Read the file name of the tsv file from the user. Assume the file name has a maximum of 25 characters.
- Open the tsv file and read the student information. Assume each last name or first name has a maximum of 25 characters.
- Compute the average exam score of each student.
- Assign a letter grade to each student based on the average exam score in the following scale:
  - A: 90 =< x
  - B: 80 =< x < 90
  - C: 70 =< x < 80
  - D: 60 =< x < 70
  - F: x < 60

- Compute the average of each exam.
- Output the last names, first names, exam scores, and letter grades of the students into a text file named report.txt. Output one student per row and separate the values with a tab character.
- Output the average of each exam, with two digits after the decimal point, at the end of report.txt. Hint: Use the precision sub-specifier to format the output.

Ex: If the input of the program is:


```
StudentInfo.tsv
```

and the contents of StudentInfo.tsv are:

```
Barrett    Edan    70  45  59
Bradshaw    Reagan  96  97  88
Charlton    Caius   73  94  80
Mayo    Tyrese  88  61  36
Stern    Brenda  90  86  45
```

the file report.txt should contain:

```
Barrett    Edan    70  45  59  F
Bradshaw    Reagan  96  97  88  A
Charlton    Caius   73  94  80  B
Mayo    Tyrese  88  61  36  D
Stern    Brenda  90  86  45  C

Averages: midterm1 83.40, midterm2 76.60, final 61.60
```


LAB ACTIVITY 7.13.1: LAB: Course Grade 0 / 10

Downloadable files

StudentInfo.tsv **Download**

main.c Load default template...

```
1  #include <stdio.h>
2
3  int main(void) {
4
5     /* TODO: Declare any necessary variables here. */
6
7
8     /* TODO: Read a file name from the user and read the tsv file here. */
```

```
9
10
11    /* TODO: Compute student grades and exam averages, then output results to a text f
12
13    return 0;
14 }
15
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

7.14 LAB: Parsing food data

This section has been set as optional by your instructor.

Given a text file containing the availability of food items, write a program that reads the information from the text file and outputs the available food items. The program first reads the name of the text file from the user. The program then reads the text file, stores the information into the four string arrays predefined in the program, and outputs the available food items in the following format: name (category) -- description

Assume the text file contains the category, name, description, and availability of at least one food item, separated by a tab character ('\t').

Hints: Use the fgets() function to read each line of the input text file. When extracting texts between the tab characters, copy the texts character-by-character until a tab character is reached. A string always ends with a null character ('\0').

Ex: If the input of the program is:


```
food.txt
```

and the contents of food.txt are:

```
Sandwiches    Ham sandwich    Classic ham sandwich    Available
Sandwiches    Chicken salad sandwich  Chicken salad sandwich  Not
available
Sandwiches    Cheeseburger    Classic cheeseburger    Not available
Salads    Caesar salad    Chunks of romaine heart lettuce dressed with
lemon juice    Available
Salads    Asian salad Mixed greens with ginger dressing, sprinkled with
sesame    Not available
Beverages    Water   16oz bottled water  Available
Beverages    Coca-Cola   16oz Coca-Cola  Not available
Mexican food    Chicken tacos   Grilled chicken breast in freshly made
tortillas    Not available
Mexican food    Beef tacos  Ground beef in freshly made tortillas
Available
Vegetarian    Avocado sandwich    Sliced avocado with fruity spread
Not available
```

the output of the program is:

```
Ham sandwich (Sandwiches) -- Classic ham sandwich
Caesar salad (Salads) -- Chunks of romaine heart lettuce dressed with
lemon juice
Water (Beverages) -- 16oz bottled water
Beef tacos (Mexican food) -- Ground beef in freshly made tortillas
```


LAB ACTIVITY | 7.14.1: LAB: Parsing food data | 0 / 10

main.c — Load default template...

```
1 #include <stdio.h>
```

```
#include <string.h>

int main(void) {
   const int MAX_LINES = 25;   // Maximum number of lines in the input text file
   const int MAX_STRING_LENGTH = 100;  // Maximum number of characters in each col
   const int MAX_LINE_LENGTH = 200;  // Maximum number of characters in each line

   // Declare 4 string arrays to store the 4 columns from the input text file
   char column1[MAX_LINES][MAX_STRING_LENGTH];
   char column2[MAX_LINES][MAX_STRING_LENGTH];
   char column3[MAX_LINES][MAX_STRING_LENGTH];
   char column4[MAX_LINES][MAX_STRING_LENGTH];

   /* Type your code here. */

   return 0;
}
```


Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

7.15 LAB: Movie show time display

This section has been set as optional by your instructor.

Write a program that reads movie data from a CSV (comma separated values) file and output the data in a formatted table. The program first reads the name of the CSV file from the user. The program then reads the CSV file and outputs the contents according to the following requirements:

- Each row contains the title, rating, and all showtimes of a unique movie.
- A space is placed before and after each vertical separator ('|') in each row.
- Column 1 displays the movie titles and is left justified with a minimum of 44 characters.
- If the movie title has more than 44 characters, output the first 44 characters only.
- Column 2 displays the movie ratings and is right justified with a minimum of 5 characters.
- Column 3 displays all the showtimes of the same movie, separated by a space.

Each row of the CSV file contains the showtime, title, and rating of a movie. Assume data of the same movie are grouped in consecutive rows.

Hints: Use the fgets() function to read each line of the input text file. When extracting texts between the commas, copy the texts character-by-character until a comma is reached. A string always ends with a null character ('\0').

Ex: If the input of the program is:

```
movies.csv
```

and the contents of movies.csv are:

```
16:40,Wonders of the World,G
20:00,Wonders of the World,G
19:00,End of the Universe,NC-17
12:45,Buffalo Bill And The Indians or Sitting Bull's History Lesson,PG
15:00,Buffalo Bill And The Indians or Sitting Bull's History Lesson,PG
19:30,Buffalo Bill And The Indians or Sitting Bull's History Lesson,PG
10:00,Adventure of Lewis and Clark,PG-13
14:30,Adventure of Lewis and Clark,PG-13
19:00,Halloween,R
```

the output of the program is:

```
Wonders of the World                          |     G | 16:40 20:00
End of the Universe                           | NC-17 | 19:00
Buffalo Bill And The Indians or Sitting Bull |    PG | 12:45 15:00
19:30
Adventure of Lewis and Clark                  | PG-13 | 10:00 14:30
Halloween                                     |     R | 19:00
```


LAB

ACTIVITY 7.15.1: LAB: Movie show time display 0 / 10

Downloadable files

movies.csv **Download**

main.c Load default template...

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void) {
6     const int MAX_TITLE_CHARS = 44;  // Maximum length of movie titles
7     const int LINE_LIMIT = 100;   // Maximum length of each line in the text file
8     char line[LINE_LIMIT];
9     char inputFileName[25];
10
11    /* Type your code here. */
12
13    return 0;
14 }
15
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **main.c** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

©zyBooks 12/22/21 12:33 1026490
Jacob Adams
UACS100Fall2021